The Simple Essence of Automatic Differentiation

CONAL ELLIOTT, Target, USA

Automatic differentiation (AD) in reverse mode (RAD) is a central component of deep learning and other uses of large-scale optimization. Commonly used RAD algorithms such as backpropagation, however, are complex and stateful, hindering deep understanding, improvement, and parallel execution. This paper develops a simple, generalized AD algorithm calculated from a simple, natural specification. The general algorithm is then specialized by varying the representation of derivatives. In particular, applying well-known constructions to a naive representation yields two RAD algorithms that are far simpler than previously known. In contrast to commonly used RAD implementations, the algorithms defined here involve no graphs, tapes, variables, partial derivatives, or mutation. They are inherently parallel-friendly, correct by construction, and usable directly from an existing programming language with no need for new data types or programming style, thanks to use of an AD-agnostic compiler plugin.

CCS Concepts: • **Mathematics of computing** → **Differential calculus**; • **Theory of computation** → *Program reasoning*; *Program specifications*;

Additional Key Words and Phrases: automatic differentiation, program calculation, category theory

ACM Reference Format:
Conal Elliott. 2018. The Simple Essence of Automatic Differentiation. *Proc. ACM Program. Lang.* 2, ICFP, Article 70 (September 2018), 29 pages. https://doi.org/10.1145/3236765

1 INTRODUCTION

Accurate, efficient, and reliable computation of derivatives has become increasingly important over the last several years, thanks in large part to the successful use of *backpropagation* in machine learning, including multi-layer neural networks, also known as "deep learning" [Goodfellow et al. 2016; Lecun et al. 2015]. Backpropagation is a specialization and independent invention of the *reverse mode* of automatic differentiation (AD) and is used to tune a parametric model to closely match observed data, using *gradient descent* (or *stochastic* gradient descent). Machine learning and other gradient-based optimization problems typically rely on derivatives of functions with very high dimensional domains and a scalar codomain—exactly the conditions under which reverse-mode AD is much more efficient than forward-mode AD (by a factor proportional to the domain dimension). Unfortunately, while forward-mode AD (FAD) is easily understood and implemented, reverse-mode AD (RAD) and backpropagation have had much more complicated explanations and implementations, involving mutation, graph construction and traversal, and "tapes" (sequences of reified, interpretable assignments, also called "traces" or "Wengert lists"). Mutation, while motivated by efficiency concerns, makes parallel execution difficult and so undermines efficiency as well. Construction and interpretation (or compilation) of graphs and tapes also add execution overhead. The importance of RAD makes its current complicated and bulky implementations especially problematic. The increasingly large machine learning (and other optimization) problems being solved with RAD (usually via backpropagation) suggest the need to find more streamlined, efficient

Author's address: Conal Elliott, conal@conal.net, Target, USA.

2475-1421/2018/9-ART70
https://doi.org/10.1145/3236765

implementations, especially with the massive hardware parallelism now readily and inexpensively available in the form of graphics processors (GPUs) and FPGAs.

Another difficulty in the practical application of AD in machine learning (ML) comes from the nature of many currently popular ML frameworks, including Caffe [Jia et al. 2014], TensorFlow [Abadi et al. 2016], and Keras [Chollet 2016]. These frameworks are designed around the notion of a "graph" (or "network") of interconnected nodes, each of which represents a mathematical operation—a sort of data flow graph. Application programs construct these graphs *explicitly*, creating nodes and connecting them to other nodes. After construction, the graphs must then be processed into a representation that is more efficient to train and to evaluate. These graphs are essentially mathematical expressions with sharing, hence directed acyclic graphs (DAGs). This paradigm of graph construction, compilation, and execution bears a striking resemblance to what programmers and compilers do all the time:

- Programs are written by a human.
- The compiler or interpreter front-end parses the program into a DAG representation.
- The compiler back-end transforms the DAGs into a form efficient for execution.
- A human runs the result of compilation.

When using a typical ML framework, programmers experience this sequence of steps *at two levels*: working with their code *and* with the graphs that their code generates. Both levels have notions of operations, variables, information flow, values, types, and parametrization. Both have execution models that must be understood.

A much simpler and cleaner foundation for ML would be to have just the programming language, omitting the graphs/networks altogether. Since ML is about (mathematical) functions, one would want to choose a programming language that supports functions well, i.e., a functional language, or at least a language with strong functional features. One might call this alternative "differentiable functional programming". In this paradigm, programmers directly define their functions of interest, using the standard tools of functional programming, with the addition of a differentiation operator (a typed higher-order function, though partial since not all computable functions are differentiable). Assuming a *purely* functional language or language subset (with simple and precise mathematical denotation), the meaning of differentiation is exactly as defined in traditional calculus.

How can we realize this vision of differentiable functional programming? One way is to create new languages, but doing so requires enormous effort to define and implement efficiently, and perhaps still more effort to evangelize. Alternatively, we might choose a suitable purely functional language like Haskell and then add differentiation. The present paper embodies the latter choice, augmenting the popular Haskell compiler GHC with a plugin that converts standard Haskell code into categorical form to be instantiated in any of a variety of categories, including differentiable functions [Elliott 2017].

This paper makes the following specific contributions:

- Beginning with a simple category of derivative-augmented functions, specify AD simply and precisely by requiring this augmentation (relative to regular functions) to be homomorphic with respect to a collection of standard categorical abstractions and primitive mathematical operations.
- Calculate a correct-by-construction AD implementation from the homomorphic specification.
- Generalizing AD by replacing linear maps (general derivative values) with an arbitrary cartesian category [Elliott 2017], define several AD variations, all stemming from different representations of linear maps: functions (satisfying linearity), "generalized matrices" (composed representable functors), continuation-based transformations of any linear map

representation, and dualized versions of any linear map representation. The latter two variations yield correct-by-construction implementations of reverse-mode AD that are much simpler than previously known and are composed from generally useful components. The choice of dualized linear functions for gradient computations is particularly compelling in simplicity. It also appears to be quite efficient—requiring no matrix-level representations or computations—and is suitable for gradient-based optimization, e.g., for machine learning. In contrast to conventional reverse-mode AD algorithms, all algorithms in this paper are free of mutation and hence naturally parallel. A similar construction yields forward-mode AD.

2 WHAT'S A DERIVATIVE?

Since automatic differentiation (AD) has to do with computing derivatives, let's begin by considering what derivatives are. If your introductory calculus class was like mine, you learned that the derivative $f'\ x$ of a function $f :: \mathbb{R} \to \mathbb{R}$ at a point x (in the domain of f) is a *number*, defined as follows:

$$f'\ x = \lim_{\varepsilon \to 0} \frac{f\ (x + \varepsilon) - f\ x}{\varepsilon} \tag{1}$$

That is, $f'\ x$ tells us how fast f is scaling input changes at x.

How well does this definition hold up beyond functions of type $\mathbb{R} \to \mathbb{R}$? It will do fine with complex numbers ($\mathbb{C} \to \mathbb{C}$), where division is also defined. Extending to $\mathbb{R} \to \mathbb{R}^n$ also works if we interpret the ratio as dividing a vector (in $\mathbb{R}^n$) by a scalar in the usual way. When we extend to $\mathbb{R}^m \to \mathbb{R}^n$ (or even $\mathbb{R}^m \to \mathbb{R}$), however, this definition no longer makes sense, as it would rely on dividing *by* a vector $\varepsilon :: \mathbb{R}^m$.

This difficulty of differentiation with non-scalar domains is usually addressed with the notion of "partial derivatives" with respect to the m scalar components of the domain $\mathbb{R}^m$, often written "$\partial f / \partial x_j$" for $j \in \{1, \ldots, m\}$. When the codomain $\mathbb{R}^n$ is also non-scalar (i.e., $n > 1$), we have a *matrix* $\mathbf{J}$ (the *Jacobian*), with $\mathbf{J}_{ij} = \partial f_i / \partial x_j$ for $i \in \{1, \ldots, n\}$, where each f_i projects out the i^{th} scalar value from the result of f.

So far, we've seen that the derivative of a function could be a single number (for $\mathbb{R} \to \mathbb{R}$), or a vector (for $\mathbb{R} \to \mathbb{R}^n$), or a matrix (for $\mathbb{R}^m \to \mathbb{R}^n$). Moreover, each of these situations has an accompanying chain rule, which says how to differentiate the composition of two functions. Where the scalar chain rule involves multiplying two scalar derivatives, the vector chain rule involves "multiplying" two *matrices* $\mathbf{A}$ and $\mathbf{B}$ (the Jacobians), defined as follows:

$$(\mathbf{A} \cdot \mathbf{B})_{ij} = \sum_{k=1}^{m} \mathbf{A}_{ik} \cdot \mathbf{B}_{kj}$$

Since one can think of scalars as a special case of vectors, and scalar multiplication as a special case of matrix multiplication, perhaps we've reached the needed generality. When we turn our attention to higher derivatives (which are derivatives of derivatives), however, the situation gets more complicated, and we need yet higher-dimensional representations, with correspondingly more complex chain rules.

Fortunately, there is a single, elegant generalization of differentiation with a correspondingly simple chain rule. First, reword Definition 1 above as follows:

$$\lim_{\varepsilon \to 0} \frac{f\ (x + \varepsilon) - f\ x}{\varepsilon} - f'\ x = 0$$

Equivalently,

$$\lim_{\varepsilon \to 0} \frac{f\ (x + \varepsilon) - (f\ x + \varepsilon \cdot f'\ x)}{\varepsilon} = 0$$

Notice that $f'\ x$ is used to linearly transform ε. Next, generalize this condition to say that $f'\ x$ is a *linear map* such that

$$\lim_{\varepsilon \to 0} \frac{\|f\ (x+\varepsilon) - (f\ x + f'\ x\ \varepsilon)\|}{\|\varepsilon\|} = 0.$$

In other words, $f'\ x$ is a *local linear approximation* of f at x. When an $f'\ x$ satisfying this condition exists, it is indeed unique [Spivak 1965, chapter 2].

The derivative of a function $f :: a \to b$ at some value in a is thus not a number, vector, matrix, or higher-dimensional variant, but rather a *linear map* (also called "linear transformation") from a to b, which we will write as "$a \multimap b$". The numbers, vectors, matrices, etc mentioned above are all different *representations* of linear maps; and the various forms of "multiplication" appearing in their associated chain rules are all implementations of linear map *composition* for those representations. Here, a and b must be vector spaces that share a common underlying field. Written as a Haskell-style type signature (but omitting vector space constraints),

$$\mathcal{D} :: (a \to b) \to (a \to (a \multimap b))$$

From the type of $\mathcal{D}$, it follows that differentiating twice has the following type:[1]

$$\mathcal{D}^2 = \mathcal{D} \circ \mathcal{D} ::\ (a \to b) \to (a \to (a \multimap a \multimap b))$$

The type $a \multimap a \multimap b$ is a linear map that yields a linear map, which is the curried form of a *bilinear* map. Likewise, differentiating k times yields a k-linear map curried $k-1$ times. For instance, the *Hessian* matrix H corresponds to the second derivative of a function $f :: \mathbb{R}^m \to \mathbb{R}$, having m rows and m columns (and satisfying the symmetry condition $H_{i,j} \equiv H_{j,i}$).

3 RULES FOR DIFFERENTIATION

3.1 Sequential Composition

With the shift to linear maps, there is one general chain rule, having a lovely form, namely that the derivative of a composition is a *composition* of the derivatives [Spivak 1965, Theorem 2-2]:

Theorem 1 (compose/"chain" rule).

$$\mathcal{D}\ (g \circ f)\ a = \mathcal{D}\ g\ (f\ a) \circ \mathcal{D}\ f\ a$$

If $f :: a \to b$ and $g :: b \to c$, then $\mathcal{D}\ f\ a :: a \multimap b$, and $\mathcal{D}\ g\ (f\ a) :: b \multimap c$, so both sides of this equation have type $a \multimap c$.[2]

Strictly speaking, Theorem 1 is not a compositional recipe for differentiating sequential compositions, i.e., it is *not* the case $\mathcal{D}\ (g \circ f)$ can be constructed solely from $\mathcal{D}\ g$ and $\mathcal{D}\ f$. Instead, it also needs f itself. Fortunately, there is a simple way to restore compositionality. Instead of constructing just the derivative of a function f, suppose we *augment* f with its derivative:

$$\mathcal{D}_0^+ :: (a \to b) \to ((a \to b) \times (a \to (a \multimap b)))\quad \text{-- first try}$$
$$\mathcal{D}_0^+\ f = (f, \mathcal{D}\ f)$$

As desired, this altered specification is compositional:

$$\begin{aligned}
&\mathcal{D}_0^+\ (g \circ f) \\
&= (g \circ f, \mathcal{D}\ (g \circ f)) && \text{-- definition of } \mathcal{D}_0^+ \\
&= (g \circ f, \lambda a \to \mathcal{D}\ g\ (f\ a) \circ \mathcal{D}\ f\ a) && \text{-- Theorem 1}
\end{aligned}$$

[1] As with "$\to$", we will take "$\multimap$" to associate rightward, so $u \multimap v \multimap w$ is equivalent to $u \multimap (v \multimap w)$.

[2] I adopt the common, if sometimes confusing, Haskell convention of sharing names between type and value variables, e.g., with a (a value variable) having type a (a type variable). Haskell value and type variable names live in different name spaces and are distinguished by syntactic context.

Note that $\mathcal{D}_0^+ (g \circ f)$ is assembled entirely from components of $\mathcal{D}_0^+ g$ and $\mathcal{D}_0^+ f$, which is to say from g, $\mathcal{D}\ g$, f, and $\mathcal{D}\ f$. Writing out $g \circ f$ as $\lambda a \to g\ (f\ a)$ underscores that the two parts of $\mathcal{D}_0^+ (g \circ f)\ a$ both involve $f\ a$. Computing these parts independently thus requires redundant work. Moreover, the chain rule itself requires applying a function and its derivative (namely f and $\mathcal{D}\ f$) to the same a. Since the chain rule gets applied recursively to nested compositions, this redundant work multiplies greatly, resulting in an impractically expensive algorithm.

Fortunately, this efficiency problem is easily fixed. Instead of pairing f and $\mathcal{D}\ f$, *combine* them:[3]

$$\mathcal{D}^+ :: (a \to b) \to (a \to b \times (a \multimap b)) \quad \text{-- better!}$$
$$\mathcal{D}^+ f\ a = (f\ a, \mathcal{D}\ f\ a)$$

Combining f and $\mathcal{D}\ f$ into a single function in this way enables us to eliminate the redundant computation of $f\ a$ in $\mathcal{D}^+ (g \circ f)\ a$, as follows:

Corollary 1.1 (See proof [Elliott 2018, Appendix C]). *$\mathcal{D}^+$ is (efficiently) compositional with respect to $(\circ)$. Specifically,*

$$\mathcal{D}^+ (g \circ f)\ a = \textbf{let}\ \{(b, f') = \mathcal{D}^+ f\ a; (c, g') = \mathcal{D}^+ g\ b\}\ \textbf{in}\ (c, g' \circ f')$$

3.2 Parallel Composition

The chain rule, telling how to differentiate sequential compositions, gets a lot of attention in calculus classes and in automatic and symbolic differentiation. There are other important ways to combine functions, however, and examining them yields additional helpful tools. Another operation (pronounced "cross") combines two functions in *parallel* [Gibbons 2002]:[4]

$$(\times) :: (a \to c) \to (b \to d) \to (a \times b \to c \times d)$$
$$f \times g = \lambda(a, b) \to (f\ a, g\ b)$$

While the derivative of a sequential composition is a sequential composition of derivatives, the derivative of a parallel composition is a parallel composition of the derivatives [Spivak 1965, variant of Theorem 2-3 (3)]:

Theorem 2 (cross rule).

$$\mathcal{D}\ (f \times g)\ (a, b) = \mathcal{D}\ f\ a \times \mathcal{D}\ g\ b$$

If $f :: a \to c$ and $g :: b \to d$, then $\mathcal{D}\ f\ a :: a \multimap c$ and $\mathcal{D}\ g\ b :: b \multimap d$, so both sides of this equation have type $a \times b \multimap c \times d$.

Theorem 2 gives us what we need to construct $\mathcal{D}^+ (f \times g)$ compositionally:

Corollary 2.1 (See proof [Elliott 2018, Appendix C]). *$\mathcal{D}^+$ is compositional with respect to $(\times)$. Specifically,*

$$\mathcal{D}^+ (f \times g)\ (a, b) = \textbf{let}\ \{(c, f') = \mathcal{D}^+ f\ a; (d, g') = \mathcal{D}^+ g\ b\}\ \textbf{in}\ ((c, d), f' \times g')$$

An important point left implicit in the discussion above is that sequential and parallel composition preserve linearity. This property is what makes it meaningful to use these forms to combine derivatives, i.e., linear maps, as we've done above.

[3] The precedence of "$\times$" is tighter than that of "$\to$" and "$\multimap$", so $a \to b \times (a \multimap b)$ is equivalent to $a \to (b \times (a \multimap b))$.
[4] By "parallel", I mean without data dependencies. Operationally, the two functions can be applied simultaneously or not.

3.3 Linear Functions

A function f is said to be *linear* when it distributes over (preserves the structure of) vector addition and scalar multiplication, i.e.,

$$f\ (a + a') = f\ a + f\ a'$$
$$f\ (s \cdot a) \quad = s \cdot f\ a$$

In addition to Theorems 1 and 2, we will want one more broadly useful rule, namely that *the derivative of every linear function is itself, everywhere* [Spivak 1965, Theorem 2-3 (2)]:

Theorem 3 (linear rule). *For all linear functions f, $\mathcal{D}\ f\ a = f$.*

This statement may sound surprising at first, but less so when we recall that the $\mathcal{D}\ f\ a$ is a local linear approximation of f at a, so we're simply saying that linear functions are their own perfect linear approximations.

For example, consider the function $id = \lambda a \to a$. Theorem 3 says that $\mathcal{D}\ id\ a = id$. When expressed via typical *representations* of linear maps, this property may be expressed as saying that $\mathcal{D}\ id\ a$ is the number one or is an identity matrix (with ones on the diagonal and zeros elsewhere). Likewise, consider the (linear) function $fst\ (a, b) = a$, for which Theorem 3 says $\mathcal{D}\ fst\ (a, b) = fst$. This property, when expressed via typical *representations* of linear maps, would appear as saying that $\mathcal{D}\ fst\ a$ comprises the partial derivatives one and zero if $a, b :: \mathbb{R}$. More generally, if $a :: \mathbb{R}^m$ and $b :: \mathbb{R}^n$, then the Jacobian matrix representation has shape $m \times (m + n)$ (i.e., m rows and $m + n$ columns) and is formed by the horizontal juxtaposition of an $m \times m$ identity matrix on the left with an $m \times n$ zero matrix on the right. This $m \times (m + n)$ matrix, however, represents $fst :: \mathbb{R}^m \times \mathbb{R}^n \multimap \mathbb{R}^m$. Note how much simpler it is to say $\mathcal{D}\ fst\ (a, b) = fst$, and with no loss of precision!

Given Theorem 3, we can construct $\mathcal{D}^+ f$ for all linear f:

Corollary 3.1. *For all linear functions f, $\mathcal{D}^+ f = \lambda a \to (f\ a, f)$. (Proof: immediate from the $\mathcal{D}^+$ definition and Theorem 3.)*

4 PUTTING THE PIECES TOGETHER

The definition of $\mathcal{D}^+$ on page 5 is a precise specification; but it is not an implementation, since $\mathcal{D}$ itself is not computable [Pour-El and Richards 1978, 1983]. Corollaries 1.1 through 3.1 provide insight into the compositional nature of $\mathcal{D}^+$ in exactly the form we can now assemble into a correct-by-construction implementation.

Although differentiation is not computable when given just an arbitrary computable function, we can instead build up differentiable functions compositionally, using exactly the forms introduced above, (namely $(\circ)$, $(\times)$ and linear functions), together with various non-linear primitives having known derivatives. Computations expressed in this vocabulary are differentiable by construction thanks to Corollaries 1.1 through 3.1. The building blocks above are not just a random assortment, but rather a fundamental language of mathematics, logic, and computation, known as *category theory* [Awodey 2006; Lawvere and Schanuel 2009; Mac Lane 1998]. While it would be unpleasant to program directly in such an austere language, its foundational nature enables instead an automatic conversion from programs written in more conventional functional languages [Elliott 2017; Lambek 1980, 1986].

4.1 Categories

The central notion in category theory is that of a *category*, comprising *objects* (generalizing sets or types) and *morphisms* (generalizing functions between sets or types). For the purpose of this paper, we will take objects to be types in our program, and morphisms to be enhanced functions.

We will introduce morphisms using Haskell-style type signatures, such as "$f :: a \rightsquigarrow b \in \mathcal{U}$", where "$\rightsquigarrow$" refers to the morphisms for a category $\mathcal{U}$, with a and b being the *domain* and *codomain* objects/types for f. In most cases, we will omit the "$\in \mathcal{U}$", where choice of category is (hopefully) clear from context. Each category $\mathcal{U}$ has a distinguished *identity* morphism $id :: a \rightsquigarrow a \in \mathcal{U}$ for every object/type a in the category. For any two morphisms $f :: a \rightsquigarrow b \in \mathcal{U}$ and $g :: b \rightsquigarrow c \in \mathcal{U}$ (note same category and matching types/objects b), there is also the composition $g \circ f :: a \rightsquigarrow c \in \mathcal{U}$. The category laws state that (a) id is the left and right identity for composition, and (b) composition is associative. You are probably already familiar with at least one example of a category, namely functions, in which id and $(\circ)$ are the identity function and function composition.

Although Haskell's type system cannot capture the category laws explicitly, we can express the two required operations as a Haskell type class, along with a familiar instance:

```
class Category k where
  id  :: a `k` a
  (∘) :: (b `k` c) → (a `k` b) → (a `k` c)
```

```
instance Category (→) where
  id = λa → a
  g ∘ f = λa → g (f a)
```

Another example is *linear* functions, which we've written "$a \multimap b$" above. Still another example is *differentiable* functions[5], which we can see by noting three facts:

- The identity function is differentiable, as witnessed by Theorem 3 and the linearity of id.
- The composition of differentiable functions is differentiable, as Theorem 1 attests.
- The category laws (identity and associativity) hold, because differentiable functions form a subset of all functions.

Each category forms its own world, with morphisms relating objects within that category. To bridge between these worlds, there are *functors*, which connect a category $\mathcal{U}$ to a (possibly different) category $\mathcal{V}$. Such a functor F maps objects in $\mathcal{U}$ to objects in $\mathcal{V}$, *and* morphisms in $\mathcal{U}$ to morphisms in $\mathcal{V}$. If $f :: a \rightsquigarrow b \in \mathcal{U}$ is a morphism, then a *functor* F from $\mathcal{U}$ to $\mathcal{V}$ transforms $f \in \mathcal{U}$ to a morphism $F\ f :: F\ a \dashrightarrow F\ b \in \mathcal{V}$, i.e., the domain and codomain of the transformed morphism $F\ f \in \mathcal{V}$ must be the transformed versions of the domain and codomain of $f \in \mathcal{U}$. In this paper, the categories use types as objects, while the functors map these types to themselves.[6] The functor must also preserve "categorical" structure:[7]

$$F\ id = id$$

$$F\ (g \circ f) = F\ g \circ F\ f$$

Crucially to the topic of this paper, Corollaries 3.1 and 1.1 say more than that differentiable functions form a category. They also point us to a new, easily implemented category, for which $\mathcal{D}^+$ is in fact a functor. This new category is simply the representation that $\mathcal{D}^+$ produces: $a \to b \times (a \multimap b)$, considered as having domain a and codomain b. The functor nature of $\mathcal{D}^+$ will be exactly what we need to in order to program in a familiar and direct way in a pleasant functional language such as Haskell and have a compiler convert to differentiable functions automatically.

To make the new category more explicit, package the result type of $\mathcal{D}^+$ in a new data type:

```
newtype D a b = D (a → b × (a ⊸ b))
```

Then adapt $\mathcal{D}^+$ to use this new data type by simply applying the D constructor to the result of $\mathcal{D}^+$:

[5] There are many examples of categories besides restricted forms of functions, including relations, logics, partial orders, and even matrices.

[6] In contrast, Haskell's functors stay within the same category and do change types.

[7] Making the categories explicit, $F\ (id \in \mathcal{U}) = (id \in \mathcal{V})$ and $F\ (g \circ f \in \mathcal{U}) = (F\ g \circ F\ f \in \mathcal{V})$.

$$\hat{\mathcal{D}} :: (a \to b) \to D\ a\ b$$
$$\hat{\mathcal{D}}\ f = D\ (\mathcal{D}^+ f)$$

Our goal is to discover a *Category* instance for D such that $\hat{\mathcal{D}}$ is a functor. This goal is essentially an algebra problem, and the desired *Category* instance is a solution to that problem. Saying that $\hat{\mathcal{D}}$ is a functor is equivalent to the following two conditions for all suitably typed functions f and g:[8]

$$id = \hat{\mathcal{D}}\ id$$

$$\hat{\mathcal{D}}\ g \circ \hat{\mathcal{D}}\ f = \hat{\mathcal{D}}\ (g \circ f)$$

Equivalently, by the definition of $\hat{\mathcal{D}}$,

$$id = D\ (\mathcal{D}^+\ id)$$

$$D\ (\mathcal{D}^+ g) \circ D\ (\mathcal{D}^+ f) = D\ (\mathcal{D}^+\ (g \circ f))$$

Now recall the following results from Corollaries 3.1 and 1.1:

$$\mathcal{D}^+\ id = \lambda a \to (id\ a, id)$$

$$\mathcal{D}^+\ (g \circ f) = \lambda a \to \textbf{let}\ \{(b, f') = \mathcal{D}^+ f\ a; (c, g') = \mathcal{D}^+ g\ b\}\ \textbf{in}\ (c, g' \circ f')$$

Then use these two facts to rewrite the right-hand sides of the functor specification for $\hat{\mathcal{D}}$:

$$id = D\ (\lambda a \to (a, id))$$

$$D\ (\mathcal{D}^+ g) \circ D\ (\mathcal{D}^+ f) = D\ (\lambda a \to \textbf{let}\ \{(b, f') = \mathcal{D}^+ f\ a; (c, g') = \mathcal{D}^+ g\ b\}\ \textbf{in}\ (c, g' \circ f'))$$

The *id* equation is trivially solvable by *defining* $id = D\ (\lambda a \to (a, id))$. To solve the $(\circ)$ equation, generalize it to a *stronger* condition:[9]

$$D\ g \circ D\ f = D\ (\lambda a \to \textbf{let}\ \{(b, f') = f\ a; (c, g') = g\ b\}\ \textbf{in}\ (c, g' \circ f'))$$

The solution of this stronger condition is immediate, leading to the following instance as a sufficient condition for $\hat{\mathcal{D}}$ being a functor:

```
linearD :: (a → b) → D a b
linearD f = D (λa → (f a, f))

instance Category D where
  id = linearD id
  D g ∘ D f = D (λa → let {(b, f') = f a; (c, g') = g b} in (c, g' ∘ f'))
```

Factoring out *linearD* will also tidy up treatment of other linear functions.

Before we get too pleased with this definition, let's remember that for D to be a category requires more than having definitions for *id* and $(\circ)$. These definitions must also satisfy the identity and composition laws. How might we go about proving that they do? Perhaps the most obvious route is take those laws, substitute our definitions of *id* and $(\circ)$, and reason equationally toward the desired conclusion. For instance, let's prove that $id \circ D\ f = D\ f$ for all $D\ f :: D\ a\ b$:[10]

```
   id ∘ D f
= D (λb → (b, id)) ∘ D f                          -- definition of id for D
= D (λa → let {(b, f') = f a; (c, g') = (b, id)} in (c, g' ∘ f'))   -- definition of (∘) for D
```

[8] The *id* and $(\circ)$ on the left-hand sides are for D, while the ones on the right are for $(\to)$.
[9] The new f is the old $\mathcal{D}^+ f$ and so has changed type from $a \to b$ to $a \to b \times (a \multimap b)$. Likewise for g.
[10] Note that *every* morphism in D has the form $D\ f$ for some f, so it suffices to consider this form.

$= D\ (\lambda a \to \textbf{let}\ \{(b, f') = f\ a\}\ \textbf{in}\ (b, id \circ f'))$ -- substitute b for c and id for g'
$= D\ (\lambda a \to \textbf{let}\ \{(b, f') = f\ a\}\ \textbf{in}\ (b, f'))$ -- $id \circ f' = f'$ (category law)
$= D\ (\lambda a \to f\ a)$ -- replace (b, f') by its definition
$= D\ f$ -- η-reduction

We can prove the other required properties similarly. Fortunately, there is a way to bypass the need for these painstaking proofs, and instead rely *only* on our original specification for this *Category* instance, namely that $\mathcal{D}^+$ is a functor. To buy this proof convenience, we have to make one concession, namely that we consider only morphisms in D that arise from $\hat{\mathcal{D}}$, i.e., only $\hat{f} :: D\ a\ b$ such that $\hat{f} = \hat{\mathcal{D}}\ f$ for some $f :: a \to b$. We can ensure that indeed only such $\hat{f}$ do arise by making $D\ a\ b$ an *abstract* type, i.e., hiding its data *constructor*. The slightly more specialized requirement of our first identity property is then $id \circ \hat{\mathcal{D}}\ f = \hat{\mathcal{D}}\ f$ for any $f :: a \to b$, which follows easily:

$id \circ \hat{\mathcal{D}}\ f$
$= \hat{\mathcal{D}}\ id \circ \hat{\mathcal{D}}\ f$ -- functor law for id (specification of $\hat{\mathcal{D}}$)
$= \hat{\mathcal{D}}\ (id \circ f)$ -- functor law for $(\circ)$
$= \hat{\mathcal{D}}\ f$ -- category law

The other identity law is proved similarly. Associativity has a similar flavor as well:

$\hat{\mathcal{D}}\ h \circ (\hat{\mathcal{D}}\ g \circ \hat{\mathcal{D}}\ f)$
$= \hat{\mathcal{D}}\ h \circ \hat{\mathcal{D}}\ (g \circ f)$ -- functor law for $(\circ)$
$= \hat{\mathcal{D}}\ (h \circ (g \circ f))$ -- functor law for $(\circ)$
$= \hat{\mathcal{D}}\ ((h \circ g) \circ f)$ -- category law
$= \hat{\mathcal{D}}\ (h \circ g) \circ \hat{\mathcal{D}}\ f$ -- functor law for $(\circ)$
$= (\hat{\mathcal{D}}\ h \circ \hat{\mathcal{D}}\ g) \circ \hat{\mathcal{D}}\ f$ -- functor law for $(\circ)$

Note how mechanical these proofs are. Each uses only the functor laws plus the particular category law on functions that corresponds to the one being proved for D. The proofs rely on nothing about the nature of D or $\hat{\mathcal{D}}$ beyond the functor laws. The importance of this observation is that we *never* need to perform these proofs when we specify category instances via a functor.

4.2 Monoidal Categories

Section 3.2 introduced parallel composition. This operation generalizes to play an important role in category theory as part of the notion of a *monoidal category*:

```
class Category k ⇒ Monoidal k where
  (×) :: (a `k` c) → (b `k` d) → ((a × b) `k` (c × d))

instance Monoidal (→) where
  f × g = λ(a, b) → (f a, g b)
```

More generally, a category k can be monoidal over constructions other than products, but cartesian products (ordered pairs) suffice for this paper.

Two monoidal categories can be related by a *monoidal functor*, which is a functor that also preserves the monoidal structure. That is, a monoidal functor F from monoidal category $\mathcal{U}$ to monoidal category $\mathcal{V}$, besides mapping objects and morphisms in $\mathcal{U}$ to counterparts in $\mathcal{V}$ while preserving the category structure (id and $(\circ)$), *also* preserves the monoidal structure:

$$F\ (f \times g) = F\ f \times F\ g$$

Just as Corollaries 1.1 and 3.1 were key to deriving a correct-by-construction *Category* instance from the specification that $\hat{\mathcal{D}}$ is a functor, Corollary 2.1 leads to a correct *Monoidal* instance from the specification that $\hat{\mathcal{D}}$ is a monoidal functor, as we'll now see.

Let F be $\hat{\mathcal{D}}$ in the reversed form of the monoidal functor equation above, and expand $\hat{\mathcal{D}}$ to its definition as $D \circ \mathcal{D}^+$:

$$D\ (\mathcal{D}^+\ f) \times D\ (\mathcal{D}^+\ g) = D\ (\mathcal{D}^+\ (f \times g))$$

By Corollary 2.1,

$$\mathcal{D}^+\ (f \times g) = \lambda(a, b) \to \textbf{let}\ \{(c, f') = \mathcal{D}^+\ f\ a; (d, g') = \mathcal{D}^+\ g\ b\}\ \textbf{in}\ ((c, d), f' \times g')$$

Now substitute the left-hand side of this equation into the right-hand side of the of the monoidal functor property for $\hat{\mathcal{D}}$, and *strengthen* the condition by generalizing from $\mathcal{D}^+\ f$ and $\mathcal{D}^+\ g$:

$$D\ f \times D\ g = D\ (\lambda(a, b) \to \textbf{let}\ \{(c, f') = f\ a; (d, g') = g\ b\}\ \textbf{in}\ ((c, d), f' \times g'))$$

This strengthened form of the specification can be converted directly to a sufficient definition:

```
instance Monoidal D where
  D f × D g = D (λ(a, b) → let {(c, f') = f a; (d, g') = g b} in ((c, d), f' × g'))
```

4.3 Cartesian Categories

The *Monoidal* abstraction provides a way to combine two functions but not separate them. It also gives no way to duplicate or discard information. These additional abilities require another algebraic abstraction, namely that of *cartesian category*, adding operations for projection and duplication:

```
class Monoidal k ⇒ Cartesian k where
  exl :: (a × b) `k` a
  exr :: (a × b) `k` b
  dup :: a `k` (a × a)
```

```
instance Cartesian (→) where
  exl = λ(a, b) → a
  exr = λ(a, b) → b
  dup = λa → (a, a)
```

Two cartesian categories can be related by a *cartesian functor*, which additionally preserves the cartesian structure. That is, a cartesian functor F from cartesian category $\mathcal{U}$ to cartesian category $\mathcal{V}$, besides mapping objects and morphisms in $\mathcal{U}$ to counterparts in $\mathcal{V}$ while preserving the category and monoidal structure (id, $(\circ)$, and $(\times)$), *also* preserves the cartesian structure:

$$\begin{aligned} F\ exl &= exl \\ F\ exr &= exr \\ F\ dup &= dup \end{aligned}$$

Just as Corollaries 1.1 through 3.1 were key to deriving a correct-by-construction *Category* and *Monoidal* instances from the specification that $\hat{\mathcal{D}}$ is a functor and a monoidal functor respectively, Corollary 3.1 enables a correct-by-construction *Cartesian* instance from the specification that $\hat{\mathcal{D}}$ is a cartesian functor. Let F be $\hat{\mathcal{D}}$ in the reversed forms of cartesian functor equations above, and expand $\hat{\mathcal{D}}$ to its definition as $D \circ \mathcal{D}^+$:

$$\begin{aligned} exl &= D\ (\mathcal{D}^+\ exl) \\ exr &= D\ (\mathcal{D}^+\ exr) \\ dup &= D\ (\mathcal{D}^+\ dup) \end{aligned}$$

Next, by Corollary 3.1, together with the linearity of *exl*, *exr*, and *dup*,

$$\begin{aligned} \mathcal{D}^+\ exl &= \lambda p \to (exl\ \ p, exl\ ) \\ \mathcal{D}^+\ exr &= \lambda p \to (exr\ \ p, exr\ ) \\ \mathcal{D}^+\ dup &= \lambda a \to (dup\ a, dup) \end{aligned}$$

Now substitute the left-hand sides of these three properties into the right-hand sides of the of the cartesian functor properties for $\hat{\mathcal{D}}$, and recall the definition of *linearD*:

$$\begin{aligned} exl &= linearD\ exl \\ exr &= linearD\ exr \\ dup &= linearD\ dup \end{aligned}$$

This form of the specification can be turned directly into a sufficient definition:

instance *Cartesian D* **where**
  $exl = linearD\ exl$
  $exr = linearD\ exr$
  $dup = linearD\ dup$

4.4 Cocartesian Categories

Cartesian categories have a dual, known as *cocartesian categories*, in which each cartesian operation has a mirror image with morphisms reversed (swapping domain and codomain) and coproducts replacing products. In general, each category can have its own notion of coproduct, e.g., sum (disjoint union) types for the $(\to)$ category. In this paper, however, coproducts will coincide with categorical products, i.e., we'll be using biproduct categories [Macedo and Oliveira 2013]:

class *Category* $k \Rightarrow$ *Cocartesian* k **where**
  $inl :: a\ `k`\ (a \times b)$
  $inr :: b\ `k`\ (a \times b)$
  $jam :: (a \times a)\ `k`\ a$

Unlike the other classes, there is no *Cocartesian* $(\to)$ instance, and fortunately we will not need such an instance below. (There is an instance when using sums instead of cartesian products for coproducts.) Instead, we can define a category $(\to^+)$ of *additive functions* that will have a *Cocartesian* instance and that we can use to represent derivatives, as shown in Figure 1. These instances rely on one more feature of the *Category* class not yet mentioned, namely an associated constraint [Bolingbroke 2011] Obj_k. In the actual class definitions, Obj_k constrains the types involved in all categorical operations.

Unsurprisingly, there is a notion of *cocartesian functor*, saying that the cocartesian structure is preserved, i.e.,

$$\begin{aligned} F\ inl &= inl \\ F\ inr &= inr \\ F\ jam &= jam \end{aligned}$$

4.5 Derived Operations

With *dup*, we can define an alternative to $(\times)$ that takes two morphisms sharing a domain:

$(\vartriangle) :: Cartesian\ k \Rightarrow (a\ `k`\ c) \to (a\ `k`\ d) \to (a\ `k`\ (c \times d))$
$f \vartriangle g = (f \times g) \circ dup$

The $(\vartriangle)$ operation is particularly useful for translating the λ-calculus to categorical form [Elliott 2017, Section 3].

Dually, *jam* lets us define a second alternative to $(\times)$ for two morphisms sharing a *codomain*:

$(\triangledown) :: Cocartesian\ k \Rightarrow (c\ `k`\ a) \to (d\ `k`\ a) \to ((c \times d)\ `k`\ a)$
$f \triangledown g = jam \circ (f \times g)$

```
newtype a ↛+ b = AddFun (a → b)

instance Category (↛+) where
  type Obj_(↛+) = Additive
  id = AddFun id
  AddFun g ∘ AddFun f = AddFun (g ∘ f)

instance Monoidal (↛+) where
  AddFun f × AddFun g = AddFun (f × g)

instance Cartesian (↛+) where
  exl = AddFun exl
  exr = AddFun exr
  dup = AddFun dup

instance Cocartesian (↛+) where
  inl = AddFun inlF
  inr = AddFun inrF
  jam = AddFun jamF

inlF :: Additive b ⇒ a → a × b
inrF :: Additive a ⇒ b → a × b
jamF :: Additive a ⇒ a × a → a

inlF = λa → (a, 0)
inrF = λb → (0, b)
jamF = λ(a, b) → a + b
```

Fig. 1. Additive functions

The $(\triangle)$ and $(\triangledown)$ operations are invertible in uncurried form [Gibbons 2002]:

```
fork   :: Cartesian   k ⇒ (a `k` c) × (a `k` d) → (a `k` (c × d))
unfork :: Cartesian   k ⇒ (a `k` (c × d)) → (a `k` c) × (a `k` d)

join   :: Cocartesian k ⇒ (c `k` a) × (d `k` a) → ((c × d) `k` a)
unjoin :: Cocartesian k ⇒ ((c × d) `k` a) → (c `k` a) × (d `k` a)
```

where

```
fork (f, g) = f △ g
unfork h = (exl ∘ h, exr ∘ h)

join (f, g) = f ▽ g
unjoin h = (h ∘ inl, h ∘ inr)
```

4.6 Numeric Operations

So far, the vocabulary we've considered comprises linear functions and combining forms $((\circ)$ and $(\times))$ that preserve linearity. To make differentiation interesting, we'll need some non-linear primitives as well. Let's now add these primitives, while continuing to derive correct implementations from simple, regular specifications in terms of homomorphisms (structure-preserving transformations). We'll define a collection of interfaces for numeric operations, roughly imitating Haskell's numeric type class hierarchy.

Haskell provides the following basic class:

```
class Num a where
  negate :: a → a
  (+), (*) :: a → a → a
  ...
```

Although this class can accommodate many different types of "numbers", the class operations are all committed to being functions. A more flexible alternative allows operations to be non-functions:

```
class NumCat k a where
  negateC :: a `k` a
  addC :: (a × a) `k` a
  mulC :: (a × a) `k` a
  ...
```

```
instance Num a ⇒ NumCat (→) a where
  negateC = negate
  addC = uncurry (+)
  mulC = uncurry (·)
  ...
```

Besides generalizing from $(\rightarrow)$ to k, we've also uncurried the operations, so as to demand less of supporting categories k. There are similar classes for other operations, such as division, powers and roots, and transcendental functions (*sin*, *cos*, *exp* etc). Note that the $(\rightarrow)$ instance uses the operations from the standard numeric classes (*Num* etc).

Differentiation rules for these operations are part of basic differential calculus:[11]

$$\mathcal{D}\ (negate\ u) = negate\ (\mathcal{D}\ u)$$
$$\mathcal{D}\ (u + v) = \mathcal{D}\ u + \mathcal{D}\ v$$
$$\mathcal{D}\ (u \cdot v) = u \cdot \mathcal{D}\ v + v \cdot \mathcal{D}\ u$$

This conventional form is unnecessarily complex, as each of these rules implicitly involves not just a numeric operation, but also an application of the chain rule. This form is also imprecise about the nature of u and v. If they are functions, then one needs to explain arithmetic on functions; and if they are not functions, then differentiation of non-functions needs explanation.

A precise and simpler presentation is to remove the arguments and talk about differentiating the primitive operations in isolation. We have the chain rule to account for context, so we do not need to involve it in every numeric operation. Since negation and (uncurried) addition are linear, we already know how to differentiate them. Multiplication is a little more involved [Spivak 1965, Theorem 2-3 (2)]:

$$\mathcal{D}\ mulC\ (a, b) = \lambda(da, db) \rightarrow da \cdot b + a \cdot db$$

Note the linearity of the right-hand side, so that the derivative of *mulC* at (a, b) for real values has the expected type: $\mathbb{R} \times \mathbb{R} \multimap \mathbb{R}$.[12] To make the linearity more apparent, and to prepare for variations later in this paper, let's now rephrase $\mathcal{D}\ mulC$ without using lambda directly. Just as *Category*, *Monoidal*, *Cartesian*, *NumCat*, etc generalize operations beyond functions, it will also be handy to generalize scalar multiplication as well:

```
class Scalable k a where
  scale :: a → (a `k` a)
```

```
instance Num a ⇒ Scalable (→+) a where
  scale a = AddFun (λda → a · da)
```

Since uncurried multiplication is bilinear, its partial application as *scale a* (for functions) is linear for all a. Now we can rephrase the product rule in terms of more general, linear language, using the derived $(\triangledown)$ operation defined in Section 4.5:

$$\mathcal{D}\ mulC\ (a, b) = scale\ b \triangledown scale\ a$$

This product rule, along with the linearity of negation and uncurried addition, enables using the same style of derivation as with operations from *Category*, *Monoidal*, and *Cartesian* above. As usual, specify the *NumCat* instance for differentiable functions by saying that $\hat{\mathcal{D}}$ preserves (*NumCat*) structure, i.e., $\hat{\mathcal{D}}\ negateC = negateC$, $\hat{\mathcal{D}}\ addC = addC$, and $\hat{\mathcal{D}}\ mulC = mulC$. Reasoning as before, we get another correct-by-construction instance for differentiable functions:

[11] The conventional differentiation rules shown here treat derivatives as numbers rather than linear maps.

[12] The derivative of uncurried multiplication generalizes to an arbitrary *bilinear* function $f :: a \times b \rightarrow c$ [Spivak 1965, Problem 2-12]: $\mathcal{D}\ f\ (a, b) = \lambda(da, db) \rightarrow f\ (da, b) + f\ (a, db)$.

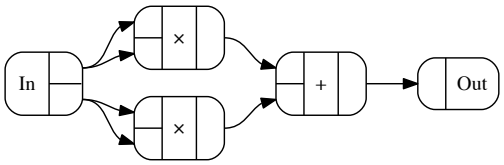

Fig. 2. *magSqr*

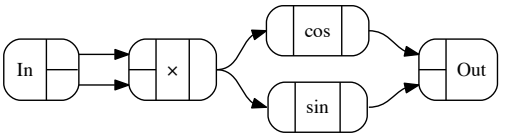

Fig. 3. *cosSinProd*

instance *NumCat D* **where**
  negateC = *linearD negateC*
  addC = *linearD addC*
  mulC = *D* (*λ*(*a*, *b*) → (*a* · *b*, *scale b* ▽ *scale a*))

Similar reasoning applies to other numeric operations, e.g.,

instance *FloatingCat D* **where**
  sinC = *D* (*λa* → (*sin a*, *scale* (*cos a*)))
  cosC = *D* (*λa* → (*cos a*, *scale* (−*sin a*)))
  expC = *D* (*λa* → **let** *e* = *exp a* **in** (*e*, *scale e*))
    ...

In what follows, the *scale* operation will play a more important role than merely tidying definitions.

5 EXAMPLES

Let's now look at some AD examples, to which we will later in the paper as well:

sqr :: *Num a* ⇒ *a* → *a*
sqr a = *a* · *a*

magSqr :: *Num a* ⇒ *a* × *a* → *a*
magSqr (*a*, *b*) = *sqr a* + *sqr b*

cosSinProd :: *Floating a* ⇒ *a* × *a* → *a* × *a*
cosSinProd (*x*, *y*) = (*cos z*, *sin z*) **where** *z* = *x* · *y*

A compiler plugin converts these definitions to categorical vocabulary [Elliott 2017]:

sqr = *mulC* ∘ (*id* △ *id*)

magSqr = *addC* ∘ (*mulC* ∘ (*exl* △ *exl*) △ *mulC* ∘ (*exr* △ *exr*))

cosSinProd = (*cosC* △ *sinC*) ∘ *mulC*

To visualize computations before differentiation, we can interpret these categorical expressions in a category of graphs [Elliott 2017, Section 7], with the results rendered in Figures 2 and 3. To see the differentiable versions, interpret these same expressions in the category of differentiable functions (*D* from Section 4.1), remove the *D* constructors to reveal the function representation, convert these functions to categorical form as well, and finally interpret the result in the graph category. The results are rendered in Figures 4 and 5. Some remarks:

- The derivatives are (linear) functions, as depicted in boxes.
- Work is shared between the function's result (sometimes called the "primal") and its derivative in Figure 5.

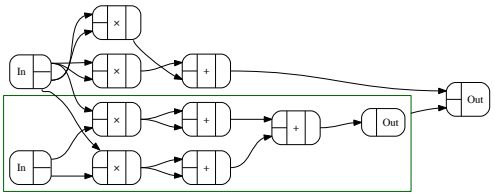

Fig. 4. $\hat{\mathcal{D}}$ *magSqr*

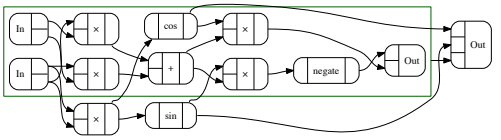

Fig. 5. $\hat{\mathcal{D}}$ *cosSinProd*

- The graphs shown here are used *solely* for visualizing functions before and after differentiation, playing no role in the programming interface or in the implementation of differentiation.

6 PROGRAMMING AS DEFINING AND SOLVING ALGEBRA PROBLEMS

Stepping back to consider what we've done, a general recipe emerges:

- Start with an expensive or even non-computable specification (here involving differentiation).
- Build the desired result into the representation of a new data type (here as the combination of a function and its derivative).
- Try to show that conversion from a simpler form (here regular functions) to the new data type—even if not computable—is *compositional* with respect to a well-understood collection of algebraic abstractions (here *Category* etc).
- If compositionality fails (as with $\mathcal{D}$, unadorned differentiation, in Section 3.1), examine the failure to find an augmented specification, iterating as needed until converging on a representation and corresponding specification that *is* compositional.
- Set up an algebra problem whose solution will be an instance of the well-understood algebraic abstraction for the chosen representation. These algebra problems always have a particular stylized form, namely that the operation being solved for is a *homomorphism* for the chosen abstractions (here including a category homomorphism, also called a "functor").
- Solve the algebra problem by using the compositionality properties.
- Rest assured that the solution satisfies the required laws, at least when the new data type is kept abstract, thanks to the homomorphic specification.

The result of this recipe is not quite an implementation of our homomorphic specification, which may after all be non-computable. Rather, it gives a computable alternative that is nearly as useful: if the input to the specified conversion is expressed in the vocabulary of the chosen algebraic abstraction, then a re-interpretation of that vocabulary in the new data type is the result of the (possibly non-computable) specification. Furthermore, if we can *automatically* convert conventionally written functional programs into the chosen algebraic vocabulary [Elliott 2017], then those programs can be re-interpreted to compute the desired specification.

7 GENERALIZING AUTOMATIC DIFFERENTIATION

Corollaries 1.1 through 3.1 all have the same form: an operation on D (differentiable functions) is defined entirely via the same operation on $(\multimap)$ (linear maps). Specifically, the sequential and parallel composition of differentiable functions rely (respectively) on sequential and parallel composition of linear maps, and likewise for each other operation. These corollaries follow closely from Theorems 1 through 3, which relate derivatives for these operations to the corresponding operations on linear maps. These properties make for a pleasantly poetic theory, but they also have a powerful, tangible benefit, which is that we can replace linear maps by any of a much broader variety of underlying categories to arrive at a greatly generalized notion of AD.

```
newtype D_k a b = D (a → b × (a `k` b))

linearD :: (a → b) → (a `k` b) → D_k a b
linearD f f' = D (λa → (f a, f'))

instance Category k ⇒ Category D_k where
  type Obj_{D_k} = Additive ∧ Obj_k
  id = linearD id id
  D g ∘ D f = D (λa → let {(b, f') = f a; (c, g') = g b} in (c, g' ∘ f'))

instance Monoidal k ⇒ Monoidal D_k where
  D f × D g = D (λ(a, b) → let {(c, f') = f a; (d, g') = g b} in ((c, d), f' × g'))

instance Cartesian k ⇒ Cartesian D_k where
  exl = linearD exl exl
  exr = linearD exr exr
  dup = linearD dup dup

instance Cocartesian k ⇒ Cocartesian D_k where
  inl = linearD inlF inl
  inr = linearD inrF inr
  jam = linearD jamF jam

instance Scalable k s ⇒ NumCat D_k s where
  negateC = linearD negateC negateC
  addC    = linearD addC    addC
  mulC    = D (λ(a, b) → (a · b, scale b ▽ scale a))
```

Fig. 6. Generalized automatic differentiation

A few small changes to the non-generalized definitions derived in Section 4 result in the generalized AD definitions shown in Figure 6:

- The new category takes as parameter a category k that replaces $(\multimap)$ in D.
- The *linearD* function takes two arrows, previously identified.
- The functionality needed of the underlying category becomes explicit.
- The constraint Obj_{D_k} is defined to be the conjunction of *Additive* (needed for the *Cocartesian* instance) and Obj_k (needed for all instances).

8 MATRICES

As an aside, let's consider matrices—the representation typically used in linear algebra—and especially the property of rectangularity. There are three (non-exclusive) possibilities for a nonempty matrix W:

- $width\ W = height\ W = 1$;
- W is the horizontal juxtaposition of two matrices U and V with $height\ W = height\ U = height\ V$, and $width\ W = width\ U + width\ V$; or

- W is the vertical juxtaposition of two matrices U and V with $width\ W = width\ U = width\ V$, and $height\ W = height\ U + height\ V$.

These three shape constraints establish and preserve rectangularity.

The vocabulary we have needed from generalized linear maps so far is exactly that of *Category*, *Cartesian*, *Cocartesian*, and *Scalable*. Let's now extract just three operations from this vocabulary:

$$
\begin{aligned}
&scale :: a \to (a\ `k`\ a)\\
&(\triangledown)\ \ :: (a\ `k`\ c) \to (b\ `k`\ c) \to ((a \times b)\ `k`\ c)\\
&(\triangle)\ \ :: (a\ `k`\ c) \to (a\ `k`\ d) \to (a\ `k`\ (c \times d))
\end{aligned}
$$

These operations exactly correspond to the three possibilities above for a nonempty matrix W, with the width and height constraints captured neatly by types. When matrices are used to represent linear maps, the domain and codomain types for the corresponding linear map are determined by the width and height of the matrix, respectively (assuming the convention of matrix on the left multiplied by a column vector on the right), together with the type of the matrix elements.

9 EXTRACTING A DATA REPRESENTATION

The generalized form of AD in Section 7 allows for different representations of linear maps (as well as alternatives to linear maps). One simple choice is to use functions, as in Figures 4 and 5. Although this choice is simple and reliable, sometimes we need a *data* representation. For instance,

- Gradient-based optimization (including in machine learning) works by searching for local minima in the domain of a differentiable function $f :: a \to s$, where a is a vector space over the scalar field s. Each step in the search is in the direction opposite of the gradient of f, which is a vector form of $\mathcal{D}\ f$.
- Computer graphics shading models rely on normal vectors. For surfaces represented in parametric form, i.e., as $f :: \mathbb{R}_2 \to \mathbb{R}_3$, normal vectors are calculated from the partial derivatives of f as vectors, which are the rows of the 3×2 Jacobian matrix that represents the derivative of f at any given point $p :: \mathbb{R}_2$.

Given a linear map $f' :: U \multimap V$ represented as a function, it is possible to extract a Jacobian matrix (including the special case of a gradient vector) by applying f' to every vector in a basis of U. A particularly convenient basis is the sequence of column vectors of an identity matrix, where the i^{th} such vector has a one in the i^{th} position and zeros elsewhere. If U has dimension m (e.g., $U = \mathbb{R}^m$), this sampling requires m passes. If m is small, then this method of extracting a Jacobian is tolerably efficient, but as dimension grows, it becomes quite expensive. In particular, many useful problems involve gradient-based optimization over very high-dimensional spaces, which is the worst case for this technique.

10 GENERALIZED MATRICES

Rather than representing derivatives as functions and then extracting a (Jacobian) matrix, a more conventional alternative is to construct and combine matrices in the first place. These matrices are usually rectangular arrays, representing $\mathbb{R}^m \multimap \mathbb{R}^n$, which interferes with the composability we get from organizing around binary cartesian products, as in the *Monoidal*, *Cartesian*, and *Cocartesian* categorical interfaces.

There is, however, an especially convenient perspective on linear algebra, known as *free vector spaces*. Given a scalar field s, any free vector space has the form $p \to s$ for some p, where the cardinality of p is the dimension of the vector space (and only finitely many p values can have non-zero images). Scaling a vector $v :: p \to s$ or adding two such vectors is defined in the usual way for functions. Rather than using functions directly as a representation, one can instead use

any representation isomorphic to such a function. In particular, we can represent vector spaces over a given field as a *representable functor*, i.e., a functor F such that $\exists p \, \forall s \; F \; s \cong p \to s$ (where "$\cong$" denotes isomorphism) This method is convenient in a richly typed functional language like Haskell, which comes with libraries of functor-level building blocks. Four such building blocks are functor product, functor composition, and their corresponding identities, which are the unit functor (containing no elements) and the identity functor (containing one element) [Magalhães et al. 2010; Magalhães et al. 2011]. One must then define the standard functionality for linear maps in the form of instances of *Category*, *Monoidal*, *Cartesian*, *Cocartesian*, and *Scalable*. Details are worked out by Elliott [2017, Section 7.4 and Appendix A]. One can use other representable functors as well, including length-typed vectors [Hermaszewski and Gamari 2017].

All of these functors give data representations of functions that save recomputation over a native function representation, as a form of functional memoization [Hinze 2000]. They also provide a composable, type-safe alternative to the more commonly used multi-dimensional arrays (often called "tensors") in machine learning libraries.

11 EFFICIENCY OF COMPOSITION

With the function representation of linear maps, composition is simple and efficient, but extracting a matrix can be quite expensive, as described in Section 9. The generalized matrix representation of Section 10 eliminates the need for this expensive extraction step but at the cost of more expensive construction operations used throughout.

One particularly important efficiency concern is that of (generalized) matrix multiplication. Although matrix multiplication is associative (because it correctly implements composition of linear maps represented as matrices), different associations can result in very different computational cost. The problem of optimally associating a chain of matrix multiplications can be solved via dynamic programming in $O(n^3)$ time [Cormen et al. 2001, Section 15.2] or in $O(n \log n)$ time with a more subtle algorithm [Hu and Shing 1981]. Solving this problem requires knowing only the sizes (heights and widths) of the matrices involved, and those sizes depend only on the types involved for a strongly typed linear map representation. One can thus choose an optimal association at compile time rather than waiting for run-time and then solving the problem repeatedly. A more sophisticated version of this question is known as the "optimal Jacobian accumulation" problem and is NP-complete [Naumann 2008].

Alternatively, for some kinds of problems we might want to choose a particular association for sequential composition. For instance, gradient-based optimization (including its use in machine learning) uses "reverse-mode" automatic differentiation (RAD), which is to say fully left-associated compositions. (Dually, "foward-mode" AD fully right-associates.) Reverse mode (including its specialization, backpropagation) is much more efficient for these problems, but is also typically given much more complicated explanations and implementations, involving mutation, graph construction, and "tapes". One of the main purposes of this paper is to demonstrate that these complications are inessential and that RAD can instead be specified and implemented quite simply.

12 REVERSE-MODE AUTOMATIC DIFFERENTIATION

The AD algorithm derived in Section 4 and generalized in Figure 6 can be thought of as a family of algorithms. For fully right-associated compositions, it becomes forward mode AD; for fully left-associated compositions, reverse-mode AD; and for all other associations, various mixed modes.

Let's now look at how to separate the associations used in formulating a differentiable function from the associations used to compose its derivatives. A practical reason for making this separation is that we want to do gradient-based optimization (calling for left association), while modular

program organization results in a mixture of compositions. Fortunately, a fairly simple technique removes the tension between efficient execution and modular program organization.

Given any category k, we can represent its morphisms by the intent to left-compose with some to-be-given morphism h. That is, represent $f :: a \mathbin{`k`} b$ by the function $(\circ f) :: (b \mathbin{`k`} r) \to (a \mathbin{`k`} r)$, where r is any object in k.[13] The morphism h will be a *continuation*, finishing the journey from f all the way to the codomain of the overall function being assembled. Building a category around this idea results in converting *all* composition patterns into fully left-associated form. This trick is akin to conversion to continuation-passing style [Appel 2007; Kennedy 2007; Reynolds 1972]. Compositions in the computation become compositions in the continuation. For instance, $g \circ f$ with a continuation k (i.e., $k \circ (g \circ f)$) becomes f with a continuation $k \circ g$ (i.e., $(k \circ g) \circ f$). The initial continuation is id (because $id \circ f = f$).

Now package up the continuation representation as a transformation from category k and codomain r to a new category, $Cont_k^r$:

$$\textbf{newtype}\ Cont_k^r\ a\ b = Cont\ ((b \mathbin{`k`} r) \to (a \mathbin{`k`} r))$$

$$cont :: Category\ k \Rightarrow (a \mathbin{`k`} b) \to Cont_k^r\ a\ b$$
$$cont\ f = Cont\ (\circ f)$$

As usual, we can derive instances for our new category by homomorphic specification:

Theorem 4 (See proof [Elliott 2018, Appendix C]). *Given the definitions in Figure 7, cont is a homomorphism with respect to each instantiated class.*

Note the pleasant symmetries in Figure 7. Each *Cartesian* or *Cocartesian* operation on $Cont_k^r$ is defined via the dual *Cocartesian* or *Cartesian* operation, together with the *join*/*unjoin* isomorphism.

The instances for $Cont_k^r$ constitute a simple algorithm for reverse-mode AD. Figures 8 and 9 show the results of $Cont_k^r$ corresponding to Figures 2 and 3 and Figures 4 and 5. The derivatives are represented as (linear) functions again, but reversed (mapping from codomain to domain).

13 GRADIENTS AND DUALITY

As a special case of reverse-mode automatic differentiation, let's consider its use to compute *gradients*, i.e., derivatives of functions with a scalar codomain, as with gradient-based optimization.

Given a vector space A over a scalar field s, the *dual* of A is $A \multimap s$, i.e., the linear maps to the underlying field [Lang 1987].[14] This dual space is also a vector space, and when A has finite dimension, it is isomorphic to its dual. In particular, every linear map in $A \multimap s$ has the form $dot\ u$ for some $u :: A$, where dot is the curried dot product:

$$\textbf{class}\ HasDot^s\ u\ \textbf{where}\ dot :: u \to (u \multimap s)$$

$$\textbf{instance}\ HasDot^{\mathbb{R}}\ \mathbb{R}\ \textbf{where}\ dot = scale$$

$$\textbf{instance}\ (HasDot^s\ a, HasDot^s\ b) \Rightarrow HasDot^s\ (a \times b)\ \textbf{where}\ dot\ (u, v) = dot\ u \triangledown dot\ v$$

The $Cont_k^r$ construction from Section 12 works for *any* type/object r, so let's take r to be the scalar field s. The internal representation of $Cont_{(\multimap)}^s\ a\ b$ is $(b \multimap s) \to (a \multimap s)$, which is isomorphic to $b \to a$. Call this representation the *dual* (or "opposite") of k:

$$\textbf{newtype}\ Dual_k\ a\ b = Dual\ (b \mathbin{`k`} a)$$

[13] Following Haskell notation for *right sections*, "$(\circ f)$" is shorthand for $\lambda h \to h \circ f$.

[14] These linear maps are variously known as "linear functionals", "linear forms", "one-forms", and "covectors".

$$\textbf{newtype}\ \mathit{Cont}^r_k\ a\ b = \mathit{Cont}\ ((b\ `k`\ r) \to (a\ `k`\ r))$$

$$\begin{aligned}
&\textbf{instance}\ \mathit{Category}\ k \Rightarrow \mathit{Category}\ \mathit{Cont}^r_k\ \textbf{where}\\
&\quad \mathit{id} = \mathit{Cont}\ \mathit{id}\\
&\quad \mathit{Cont}\ g \circ \mathit{Cont}\ f = \mathit{Cont}\ (f \circ g)
\end{aligned}$$

$$\begin{aligned}
&\textbf{instance}\ \mathit{Monoidal}\ k \Rightarrow \mathit{Monoidal}\ \mathit{Cont}^r_k\ \textbf{where}\\
&\quad \mathit{Cont}\ f \times \mathit{Cont}\ g = \mathit{Cont}\ (\mathit{join} \circ (f \times g) \circ \mathit{unjoin})
\end{aligned}$$

$$\begin{aligned}
&\textbf{instance}\ \mathit{Cartesian}\ k \Rightarrow \mathit{Cartesian}\ \mathit{Cont}^r_k\ \textbf{where}\\
&\quad \mathit{exl}\ = \mathit{Cont}\ (\mathit{join} \circ \mathit{inl})\\
&\quad \mathit{exr}\ = \mathit{Cont}\ (\mathit{join} \circ \mathit{inr})\\
&\quad \mathit{dup} = \mathit{Cont}\ (\mathit{jam} \circ \mathit{unjoin})
\end{aligned}$$

$$\begin{aligned}
&\textbf{instance}\ \mathit{Cocartesian}\ k \Rightarrow \mathit{Cocartesian}\ \mathit{Cont}^r_k\ \textbf{where}\\
&\quad \mathit{inl}\ = \mathit{Cont}\ (\mathit{exl} \circ \mathit{unjoin})\\
&\quad \mathit{inr}\ = \mathit{Cont}\ (\mathit{exr} \circ \mathit{unjoin})\\
&\quad \mathit{jam} = \mathit{Cont}\ (\mathit{join} \circ \mathit{dup})
\end{aligned}$$

$$\begin{aligned}
&\textbf{instance}\ \mathit{Scalable}\ k\ a \Rightarrow \mathit{Scalable}\ \mathit{Cont}^r_k\ a\ \textbf{where}\\
&\quad \mathit{scale}\ s = \mathit{Cont}\ (\mathit{scale}\ s)
\end{aligned}$$

Fig. 7. Continuation category transformer (specified by functoriality of *cont*)

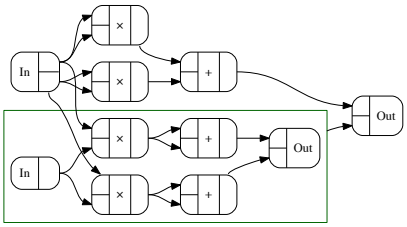

Fig. 8. *magSqr* in $D_{\mathit{Cont}^{\mathbb{R}}_{(\to^+)}}$

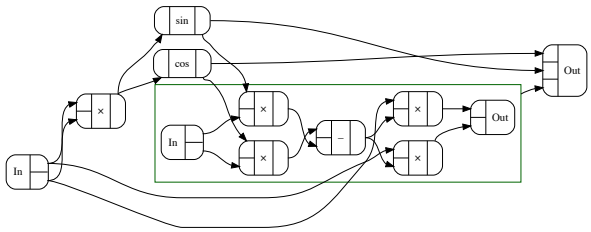

Fig. 9. *cosSinProd* in $D_{\mathit{Cont}^{\mathbb{R}}_{(\to^+)}}$

To construct dual representations of (generalized) linear maps, it suffices to convert from Cont^s_k to Dual_k by a functor we will now derive. Composing this new functor with $\mathit{cont} :: (a\ `k`\ b) \to \mathit{Cont}^s_k\ a\ b$ will give us a functor from k to Dual_k. The new to-be-derived functor:

$$\begin{aligned}
&\mathit{asDual} :: (\mathit{HasDot}^s\ a, \mathit{HasDot}^s\ b) \Rightarrow \mathit{Cont}^s_k\ a\ b \to \mathit{Dual}_k\ a\ b\\
&\mathit{asDual}\ (\mathit{Cont}\ f) = \mathit{Dual}\ (\mathit{onDot}\ f)
\end{aligned}$$

where *onDot* uses both halves of the isomorphism between $a \multimap s$ and a:

$$\begin{aligned}
&\mathit{onDot} :: (\mathit{HasDot}^s\ a, \mathit{HasDot}^s\ b) \Rightarrow ((b \multimap s) \to (a \multimap s)) \to (b \multimap a)\\
&\mathit{onDot}\ f = \mathit{dot}^{-1} \circ f \circ \mathit{dot}
\end{aligned}$$

As usual, we can derive instances for our new category by homomorphic specification:

```
instance Category k ⇒ Category Dual_k where
  id = Dual id
  Dual g ∘ Dual f = Dual (f ∘ g)

instance Monoidal k ⇒ Monoidal Dual_k where
  Dual f × Dual g = Dual (f × g)

instance Cartesian k ⇒ Cartesian Dual_k where
  exl  = Dual inl
  exr  = Dual inr
  dup = Dual jam

instance Cocartesian k ⇒ Cocartesian Dual_k where
  inl  = Dual exl
  inr  = Dual exr
  jam = Dual dup

instance Scalable k ⇒ Scalable Dual_k where
  scale s = Dual (scale s)
```

Fig. 10. Dual category transformer (specified by functoriality of *asDual*)

Theorem 5 (See proof [Elliott 2018, Appendix C]). *Given the definitions in Figure 10, asDual is a homomorphism with respect to each instantiated class.*

Note that the instances in Figure 10 exactly dualize a computation, reversing sequential compositions and swapping corresponding *Cartesian* and *Cocartesian* operations. Likewise for the derived operations:

Corollary 5.1 (See proof [Elliott 2018, Appendix C]). *The* $(\triangle)$ *and* $(\triangledown)$ *operations mutually dualize:*

$$Dual\ f \triangle Dual\ g = Dual\ (f \triangledown g)$$
$$Dual\ f \triangledown Dual\ g = Dual\ (f \triangle g)$$

Recall from Section 8, that *scale* forms 1×1 matrices, while $(\triangledown)$ and $(\triangle)$ correspond to horizontal and vertical juxtaposition, respectively. Thus, from a matrix perspective, duality is *transposition*, turning an $m \times n$ matrix into an $n \times m$ matrix. Note, however, that $Dual_k$ involves no actual matrix computations unless k does. In particular, we can simply use the category of linear functions $(\multimap)$.

Figures 11 and 12 show the results of reverse-mode AD via $D_{Dual_{(\multimap)}}$. Compare Figure 11 with Figures 4 and 8.

14 FORWARD-MODE AUTOMATIC DIFFERENTIATION

It may be interesting to note that we can turn the *Cont* and *Dual* techniques around to yield category transformers that perform full *right-* instead of left-association, converting the general, mode-independent algorithm into forward mode, thus yielding an algorithm preferable for low-dimensional domains (rather than codomains):

```
newtype Begin^r_k a b = Begin ((r `k` a) → (r `k` b))
```

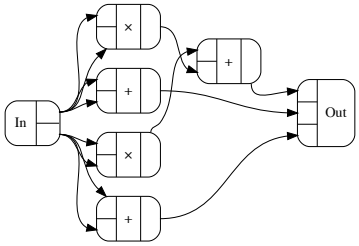

Fig. 11. *magSqr* in $D_{Dual_{(\multimap)}}$

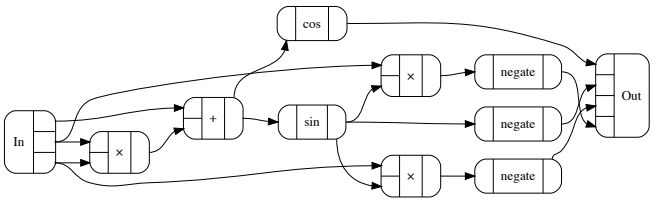

Fig. 12. $\lambda((x, y), z) \to cos\ (x + y \cdot z)$ in $D_{Dual_{(\multimap)}}$

$$begin :: Category\ k \Rightarrow (a\ `k`\ b) \to Begin^r_k\ a\ b$$
$$begin\ f = Begin\ (f \circ)$$

As usual, we can derive instances for our new category by homomorphic specification (for *begin*). Then choose r to be the scalar field s, as in Section 13, noting that $(s \multimap a) \cong a$.

15 SCALING UP

So far, we have considered binary products. Practical applications, including machine learning and other optimization problems, often involve very high-dimensional spaces. While those spaces can be encoded as nested binary products, doing so would result in unwieldy representations and prohibitively long compilation and execution times. A practical alternative is to consider n-ary products, for which we can again use representable functors. To construct and consume these "indexed" (bi)products, we'll need an indexed variant of *Monoidal*, replacing the two arguments to $(\times)$ by a (representable) functor h of morphisms:

```
class Category k ⇒ MonoidalI k h where
  crossI :: h (a `k` b) → (h a `k` h b)

instance Zip h ⇒ MonoidalI (→) h where
  crossI = zipWith id
```

Note that the collected morphisms must all agree in domain and codomain. While not required for the general categorical notion of products, this restriction accommodates Haskell's type system and seems adequate in practice so far.

Where the *Cartesian* class has two projection methods and a duplication method, the indexed counterpart has a collection of projections and one replication method:[15]

```
class MonoidalI k h ⇒ CartesianI k h where
  exI  :: h (h a `k` a)
  replI :: a `k` h a

instance (Representable h, Zip h, Pointed h) ⇒ CartesianI (→) h where
  exI  = tabulate (flip index)
  replI = point
```

Dually, where the *Cocartesian* class has two injection methods and a binary combination method, the indexed counterpart has a collection of injections and one collection-combining method:

```
class MonoidalI k h ⇒ CocartesianI k h where
  inI  :: h (a `k` h a)
  jamI :: h a `k` a
```

[15] The *index* and *tabulate* functions convert from functor to function and back [Kmett 2011].

```
instance (MonoidalI k h, Zip h) ⇒ MonoidalI D_k h where
  crossI fs = D (second crossI ∘ unzip ∘ crossI (fmap unD fs))

instance (CartesianI (→) h, CartesianI k h, Zip h) ⇒ CartesianI D_k h where
  exI   = linearD exI exI
  replI = zipWith linearD replI replI

instance (CocartesianI k h, Zip h) ⇒ CocartesianI D_k h where
  inI   = zipWith linearD inIF inI
  jamI = linearD sum jamI

unD :: D a b → (a → (b × (a ⊸ b)))
unD (D f) = f

unzip :: Functor h ⇒ h (a × b) → h a × h b
unzip = fmap exl △ fmap exr

second :: Monoidal k ⇒ (b `k` d) → ((a × b) `k` (a × d))
second g = id × g

inIF :: (Additive a, Foldable h) ⇒ h (a → h a)
inIF = tabulate (λi a → tabulate (λj → if i = j then a else 0))

class Zip h where zipWith :: (a → b → c) → h a → h b → h c
```

Fig. 13. AD for indexed biproducts

There are also indexed variants of the derived operations ($\triangle$) and ($\triangledown$) from Section 4.5:

```
forkI :: CartesianI k h ⇒ h (a `k` b) → (a `k` h b)
forkI fs = crossI fs ∘ replI

unforkF :: CartesianI k h ⇒ (a `k` h b) → h (a `k` b)
unforkF f = fmap (∘ f) exI

joinI :: CartesianI k h ⇒ h (b `k` a) → (h b `k` a)
joinI fs = jamI ∘ crossI fs

unjoinPF :: CocartesianI k h ⇒ (h b `k` a) → h (b `k` a)
unjoinPF f = fmap (f ∘) inI
```

As usual, we can derive instances by homomorphic specification:

Theorem 6 (See proof [Elliott 2018, Appendix C]). *Given the definitions in Figure 13, $\hat{\mathcal{D}}$ is a homomorphism with respect to each instantiated class.*

These indexed operations are useful in themselves but can be used to derive other operations. For instance, note the similarity between the types of *crossI* and *fmap*:

```
instance (Zip h, Additive₁ h) ⇒ MonoidalI (⊸⁺) h where
  crossI = AddFun ∘ crossI ∘ fmap unAddFun

instance (Representable h, Zip h, Pointed h, Additive₁ h) ⇒ CartesianI (⊸⁺) h where
  exI = fmap AddFun exI
  replI = AddFun replI

instance (Foldable h, Additive₁ h) ⇒ CocartesianI (⊸⁺) h where
  inI  = fmap AddFun inIF
  jamI = AddFun sum
```

Fig. 14. Indexed instances for $(\multimap^+)$

```
instance (MonoidalI k h, Functor h, Additive₁ h) ⇒ MonoidalI (Dual k) h where
  crossI = Dual ∘ crossI ∘ fmap unDual

instance (CocartesianI k h, Functor h, Additive₁ h) ⇒ CartesianI (Dual k) h where
  exI  = fmap Dual inI
  replI = Dual jamI

instance (CartesianI k h, Functor h, Additive₁ h) ⇒ CocartesianI (Dual k) h where
  inI  = fmap Dual exI
  jamI = Dual replI
```

Fig. 15. Indexed instances for $Dual_k$

```
crossI :: Monoidal h ⇒ h (a → b) → (h a → h b)
fmap   :: Functor  h ⇒   (a → b) → (h a → h b)
```

In fact, the following relationship holds: $fmap = crossI \circ replI$. This equation, together with the differentiation rules for *crossI*, *replI*, and $(\circ)$ determines differentiation for $fmap\ f$.

As with Figure 6, the operations defined in Figure 13 rely on corresponding operations for the category parameter k. Fortunately, all of those operations are linear or preserve linearity, so they can all be defined on the various representations of derivatives (linear maps) used for AD in this paper. Figures 14 and 15 show instances for two of the linear map representations defined in this paper, with the third left as an exercise for the reader. (The constraint $Additive_1\ h$ means that $\forall a.Additive\ a \Rightarrow Additive\ (h\ a)$.)

16 RELATED WORK

The literature on automatic differentiation is vast, beginning with forward mode [Wengert 1964] and later reverse mode [Rall 1981; Speelpenning 1980], with many developments since [Griewank 1989; Griewank and Walther 2008]. While most techniques and uses of AD have been directed at imperative programming, there are also variations for functional programs [Elliott 2009; Karczmarczuk 1999, 2000, 2001; Pearlmutter and Siskind 2007, 2008]. The work in this paper differs in being phrased at the level of functions/morphisms and specified by functoriality, without any mention or

manipulation of graphs or other syntactic representations.[16] Moreover, the specifications in this paper are simple enough that the various forms of AD presented can be calculated into being [Bird and de Moor 1996; Oliveira 2018], and so are correct by construction [Elliott 2018].

Pearlmutter and Siskind [2008] make the following observation:

> In this context, reverse-mode AD refers to a particular construction in which the primal data-flow graph is transformed to construct an adjoint graph that computes the sensitivity values. In the adjoint, the direction of the data-flow edges are reversed; addition nodes are replaced by fanout nodes; fanout nodes are replaced by addition nodes; and other nodes are replaced by multiplication by their linearizations. The main constructions of this paper can, in this context, be viewed as a method for constructing scaffolding that supports this adjoint computation.

The *Cont* and *Dual* category transformers described in Sections 12 and 13 (shown in Figures 7 and 10) above explain this "adjoint graph" construction without involving graphs. Data-flow edge reversal corresponds to the reversal of $(\circ)$ (from *Category*), while fanout and addition correspond to *dup* and *jam* (from *Cartesian* and *Cocartesian* respectively), which are mutually dual. Pearlmutter and Siskind [2008] further remark:

> The main technical difficulty to be faced is that reverse-mode AD must convert fanout (multiple use of a variable) in the untransformed code into addition in the reverse phase of the transformed code. We address this by expressing all straight-line code segments in A-normal form, which makes fanout lexically apparent.

The categorical approach in this paper also makes fanout easily apparent, as occurrences of *dup*, which are produced during translation from Haskell to categorical form [Elliott 2017] (via $(\vartriangle)$ as defined in Section 4.5 above). This translation is specified and implemented independently of AD and so presents no additional complexity.

Closely related to our choice of derivatives as linear maps and their categorical generalizations is the work of Macedo and Oliveira [2013], also based on biproducts (though not addressing differentiation). That work uses natural numbers as categorical objects to capture the dimensions of vectors and matrices, while the current paper uses vector spaces themselves. The difference is perhaps minor, however, since natural numbers can be thought of as representing finite sets (of corresponding cardinality), which are *bases* of finite-dimensional free vector spaces (as in Section 10). On the other hand, the duality-based gradient algorithm of Section 13 involves no matrices at all in their traditional representation (arrays of numbers) or generalized sense of Section 10 (representable functors).

Also sharing a categorical style is the work of Fong et al. [2017], formulating the backpropropagation algorithm as a functor. That work, which also uses biproducts (in monoidal but not cartesian form), does not appear to be separable from the application to machine learning, and so would seem to complement this paper. Backpropagation is a specialization of reverse-mode AD to the context of machine learning, discovered by Linnainmaa [1970] and popularized by Rumelhart et al. [1988].

The continuation transformation of Section 12 was inspired by Mitch Wand's work on continuation-based program transformation [Wand 1980]. He derived a variety of algorithms based on a single elegant technique: transform a simple recursive program into continuation-passing form, examine

[16] Of course the Haskell compiler itself manipulates syntax trees, and the compiler plugin that converts Haskell code to categorical form helps do so, but both are entirely domain-independent, with no knowledge of or special support for differentiation or linear algebra [Elliott 2017].

the continuations that arise, and find a data (rather than function) representation for them. Each such representation is a monoid, with its identity and associative operation corresponding to identity and composition of the continuations. Monoids are categories with only one object, but the technique extends to general categories. Cayley's theorem for groups (or monoids) captures this same insight and is a corollary (in retrospect) of the Yoneda lemma [Riehl 2016, Section 2.2]. The idea of using data representations for functions ("defunctionalization") was pioneered by Reynolds [1972] and further explored by Danvy and Nielsen [2001].

The notion of derivatives as linear maps is the basis of calculus on manifolds [Spivak 1965] and was also used for AD by Elliott [2009]. The latter addressed only forward-mode AD but also included all orders of derivatives.

While there are many forward-mode AD libraries for Haskell, reverse mode (RAD) has been much more difficult. The most successful implementation appears to be in the *ad* library [Kmett et al. 2010]. One RAD implementation in that library uses stable names [Peyton Jones et al. 1999] and reification [Gill 2009] to recover sharing information. Another maintains a Wengert list (or "tape") with the help of a reflection library [Kiselyov and Shan 2004]. Both implementations rely on hidden, carefully crafted use of side effects.

Chris Olah [2015] shared a vision for "differentiable functional programming" similar to that in Section 1. He pointed out that most of the patterns now used in machine learning are already found in functional programming:

> These neural network patterns are just higher order functions—that is, functions which take functions as arguments. Things like that have been studied extensively in functional programming. In fact, many of these network patterns correspond to extremely common functions, like fold. The only unusual thing is that, instead of receiving normal functions as arguments, they receive chunks of neural network.

The current paper carries this perspective further, suggesting that the essence is *differentiable functions*, with "networks" (graphs) being an unnecessary (and apparently unwise) operational choice.

This paper builds on a compiler plugin that translates Haskell programs into categorical form to be specialized to various specific categories, including differentiable functions [Elliott 2017]. (The plugin knows nothing about any specific category, including differentiable functions.) Another instance of generalized AD given there is automatic incremental evaluation of functional programs. Relative to that work, the new contributions are the $Cont^r_k$ and $Dual_k$ categories, their use to succinctly implement reverse-mode AD (by instantiating the generalized differentiation category), the precise specification of instances for D, $Cont^r_k$, and $Dual_k$ via functoriality, and the calculation of implementations from these specifications.

The implementations in this paper are quite simple and appear to be efficient as well. For instance, the duality-based version (Section 13) involves no matrices. Moreover, typical reverse-mode AD (RAD) implementations use mutation to incrementally update derivative contributions from each *use* of a variable or intermediate computation, holding onto all of these accumulators until the very end of the derivative computation. For this reason, such implementations tend to use considerable memory. In contrast, the implementations in this paper (Sections 12 and 13) are free of mutation and can easily free (reuse) memory as they run, keeping memory use low. Given the prominent use of AD, particularly with large data, performance is crucial, so it will be worthwhile to examine and compare time and space use in detail. Lack of mutation also makes the algorithms in this paper naturally parallel, potentially leading to considerable speed improvement, especially when using the functor-level (bulk) vocabulary in Section 15.

17 CONCLUSIONS

This paper develops a simple, mode-independent algorithm for automatic differentiation (AD) (Section 4), calculated from a simple, natural specification in terms of elementary category theory (functoriality). It then generalizes the algorithm, replacing linear maps (as derivatives) by an arbitrary biproduct category (Figure 6). Specializing this general algorithm to two well-known categorical constructions (Figures 7 and 10)—also calculated—yields reverse-mode AD (RAD) for general derivatives and for gradients. These RAD implementations are far simpler than previously known. In contrast to common approaches to AD, the new algorithms involve no graphs, tapes, variables, partial derivatives, or mutation, and are usable directly from an existing programming language with no need for new data types or programming style (thanks to use of an AD-agnostic compiler plugin). Only the simple essence remains.

Future work includes detailed performance analysis (compared with backpropagation and other conventional AD algorithms); efficient higher-order differentiation; and applying generalized AD to derivative-like notions, including subdifferentiation [Rockafellar 1966] and automatic incrementalization (continuing previous work [Elliott 2017]).

AD is typically said to be about the chain rule for sequential composition (Theorem 1). This paper rounds out the story with two more rules: one for parallel composition and one for all linear operations (Theorems 2 and 3). Parallel composition is usually left implicit in the special-case treatment of a collection of non-unary operations, such as addition, multiplication, division, and dot products. With explicit, general support for parallel composition, all operations come to be on equal footing, regardless of arity (as illustrated in Figure 6).

AD is also typically presented in opposition to symbolic differentiation (SD), with the latter described as applying differentiation rules symbolically. The main criticism of SD is that it can blow up expressions, resulting a great deal of redundant work. Secondly, SD requires implementation of symbolic manipulation as in a computer algebra system. In contrast, AD is described as a numeric method and can retain the complexity of the original function (within a small constant factor) if carefully implemented, as in reverse mode. The approach explored in this paper suggests a different perspective: automatic differentiation *is* symbolic differentiation performed by a compiler. Compilers already work symbolically and already take care to preserve sharing in computations, addressing both criticisms of SD.

The specification and implementation of AD in a simple, correct-by-construction, and apparently efficient manner, together with its use from a typed functional language (here via a compiler plugin), make a step toward the vision of differentiable functional programming for machine learning and other uses, as outlined in Section 1. Programmers then define their functions just as they are accustomed, differentiating where desired, without the intrusion of operational notions such as graphs with questionably defined, extralinguistic semantics.

In retrospect, two key principles enable the results in this paper:

(1) Focus on abstract notions (specified denotationally and/or axiomatically) rather than particular representations (here, derivatives as linear maps rather than as matrices). Then transform a correct, naive representation into subtler, more efficient representations.
(2) Capture the main concepts of interest directly, as first-class values (here, differentiable functions).

The second principle leads us into a quandary, because most programming languages (including Haskell) are much better suited to expressing regular computable functions than other function-like things, and yet the main AD concept is exactly a function-like thing (differentiable functions). This imbalance in suitability stems from built-in language support for functions—such as lambda, application, and variables—and would seem to explain two common strategies in AD: use of explicit

graph representations (complicating use and implementation), and overloading numeric operators (abandoning the second principle, encouraging *forward* mode AD, and leading to incorrect nested differentiation [Siskind and Pearlmutter 2008]). Fortunately, we can instead extend the notational convenience of functions to other function-like things by writing in a conventional functional language and automatically translating to other categories [Elliott 2017].

REFERENCES

Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 265–283.

Andrew W. Appel. 2007. *Compiling with Continuations*. Cambridge University Press.

Steve Awodey. 2006. *Category theory*. Oxford Logic Guides, Vol. 49. Oxford University Press.

Richard Bird and Oege de Moor. 1996. *The Algebra of Programming*. Prentice-Hall.

Max Bolingbroke. 2011. Constraint kinds for GHC. Blog post. http://blog.omega-prime.co.uk/2011/09/10/constraint-kinds-for-ghc/.

François Chollet. 2016. Keras resources. GitHub repository. https://github.com/fchollet

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2001. *Introduction to Algorithms, Third Edition*. The MIT Press and McGraw-Hill Book Company.

Olivier Danvy and Lasse R. Nielsen. 2001. Defunctionalization at work. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP '01)*. 162–174.

Conal Elliott. 2009. Beautiful differentiation. In *International Conference on Functional Programming (ICFP)*.

Conal Elliott. 2017. Compiling to categories. *Proceedings of the ACM on Programming Languages* 1, ICFP, Article 48 (Sept. 2017), 24 pages.

Conal Elliott. 2018. The simple essence of automatic differentiation (Extended version). *CoRR* abs/1804.00746 (2018).

Brendan Fong, David I. Spivak, and Rémy Tuyéras. 2017. Backprop as functor: A compositional perspective on supervised learning. *CoRR* abs/1711.10455 (2017).

Jeremy Gibbons. 2002. Calculating functional programs. In *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*. Lecture Notes in Computer Science, Vol. 2297. Springer-Verlag.

Andy Gill. 2009. Type-safe observable sharing in Haskell. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell (Haskell '09)*.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press.

Andreas Griewank. 1989. On Automatic Differentiation. In *In Mathematical Programming: Recent Developments and Applications*.

Andreas Griewank and Andrea Walther. 2008. *Evaluating Derivatives. Principles and Techniques of Algorithmic Differentiation* (second ed.). Society for Industrial and Applied Mathematics.

Joe Hermaszewski and Ben Gamari. 2017. vector-sized. http://github.com/expipiplus1/vector-sized Haskell library.

Ralf Hinze. 2000. Memo functions, polytypically!. In *2nd Workshop on Generic Programming*. 17–32.

T. C. Hu and M. T. Shing. 1981. *Computation of matrix chain products, Part I, Part II.* Technical Report STAN-CS-TR-81-875. Stanford University, Department of Computer Science.

Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross B. Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional architecture for fast feature embedding. *CoRR* abs/1408.5093 (2014).

Jerzy Karczmarczuk. 1999. Functional coding of differential forms. In *Scottish Workshop on Functional Programming*.

Jerzy Karczmarczuk. 2000. Adjoint codes in functional framework.

Jerzy Karczmarczuk. 2001. Functional differentiation of computer programs. *Higher-Order and Symbolic Computation* 14, 1 (2001).

Andrew Kennedy. 2007. Compiling with continuations, continued. In *ACM SIGPLAN International Conference on Functional Programming*.

Oleg Kiselyov and Chung-chieh Shan. 2004. Functional pearl: Implicit configurations—or, type classes reflect the values of types. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell (Haskell '04)*.

Edward Kmett. 2011. The *adjunctions* package. https://hackage.haskell.org/package/adjunctions. Haskell library.

Edward Kmett, Barak Pearlmutter, and Jeffrey Mark Siskind. 2010. The *ad* package. https://hackage.haskell.org/package/ad. Haskell library.

Joachim Lambek. 1980. From λ-calculus to cartesian closed categories. In *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, J.P. Seldin and J.R. Hindley (Eds.). Academic Press.

Joachim Lambek. 1986. Cartesian closed categories and typed lambda-calculi. In *Thirteenth Spring School of the LITP on Combinators and Functional Programming Languages*. 136–175.

Serge Lang. 1987. *Linear Algebra* (3rd ed.). Springer-Verlag.

F. William Lawvere and Stephen H. Schanuel. 2009. *Conceptual Mathematics: A First Introduction to Categories* (2nd ed.). Cambridge University Press.

Yann Lecun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *Nature* 521, 7553 (27 5 2015), 436–444.

Seppo Linnainmaa. 1970. *The representation of the cumulative rounding error of an algorithm as a Taylor expansion of the local rounding errors.* Master's thesis. University of Helsinki.

Saunders Mac Lane. 1998. *Categories for the Working Mathematician.* Springer New York.

Hugo Daniel Macedo and José Nuno Oliveira. 2013. Typing linear algebra: A biproduct-oriented approach. *Science of Computer Programming* 78, 11 (2013), 2160–2191.

José Pedro Magalhães, Atze Dijkstra, Johan Jeuring, and Andres Löh. 2010. A generic deriving mechanism for Haskell. In *Haskell Symposium*. 37–48.

José Pedro Magalhães et al. 2011. GHC.Generics. https://wiki.haskell.org/GHC.Generics Haskell wiki.

Uwe Naumann. 2008. Optimal Jacobian accumulation is NP-complete. *Mathematical Programming* 112 (2008), 427–441.

Chris Olah. 2015. Neural networks, types, and functional programming. Blog post. http://colah.github.io/posts/2015-09-NN-Types-FP/.

José Nuno Oliveira. 2018. *Program Design by Calculation.* Draft of textbook in preparation.

Barak A. Pearlmutter and Jeffrey Mark Siskind. 2007. Lazy multivariate higher-order forward-mode AD. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '07)*.

Barak A. Pearlmutter and Jeffrey Mark Siskind. 2008. Reverse-mode AD in a functional framework: Lambda the ultimate backpropagator. *ACM TOPLAS* 30, 2 (March 2008).

Simon L. Peyton Jones, Simon Marlow, and Conal Elliott. 1999. Stretching the storage manager: Weak pointers and stable names in Haskell. In *Implementation of Functional Languages*.

Marian Boykan Pour-El and Ian Richards. 1978. Differentiability properties of computable functions—A summary. *Acta Cybernetica* 4, 1 (1978), 123–125.

Marian Boykan Pour-El and Ian Richards. 1983. Computability and noncomputability in classical analysis. *Transactions of the American Mathematical Society* 275, 2 (1983), 539–560.

Louis B. Rall. 1981. *Automatic Differentiation: Techniques and Applications.* Springer-Verlag.

John C. Reynolds. 1972. Definitional interpreters for higher-order programming languages. In *Reprinted from the proceedings of the 25th ACM National Conference*. ACM, 717–740.

Emily Riehl. 2016. *Category Theory in Context.* Dover Publications.

R. Tyrrell Rockafellar. 1966. Characterization of the subdifferentials of convex functions. *Pacific Journal of Mathematics* 17, 3 (1966), 497–510.

David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. 1988. Learning representations by back-propagating errors. In *Neurocomputing: Foundations of Research*. MIT Press.

Jeffrey Mark Siskind and Barak A. Pearlmutter. 2008. Nesting forward-mode AD in a functional framework. *Higher Order Symbolic Computation* 21, 4 (2008), 361–376.

Bert Speelpenning. 1980. *Compiling fast partial derivatives of functions given by algorithms.* Ph.D. Dissertation. University of Illinois at Urbana-Champaign.

Michael Spivak. 1965. *Calculus on Manifolds: A Modern Approach to Classical Theorems of Advanced Calculus.* Addison-Wesley.

Mitchell Wand. 1980. Continuation-based program transformation strategies. *Journal of the ACM* 27, 1 (1980), 164–180.

R. E. Wengert. 1964. A simple automatic derivative evaluation program. *Communications of the ACM* 7, 8 (1964), 463–464.